Resource-agnostic Programming of Heterogeneous Architectures with Single Assignment C (SAC)

Clemens Grelck

Universiteit van Amsterdam

(Joint work with Miguel Diogo, MSc, and the whole SAC Team)

4th Distributed ASCI Supercomputer Workshop
Delft, Netherlands
February 13, 2013

The Trend on DAS and beyond: Heterogeneity

DAS-1 (1997)

- 200-Mhz Pentium Pro nodes

DAS-2 (2002)

- Dual 1-GHz Pentium III nodes

DAS-3 (2007)

- Dual AMD-Opteron nodes
- Some single core, some dual core
- 2.2-GHz 2.4-GHz 2.6-GHz

DAS-4 (2011)

?

Computing in the Age of Multi- and Many-Core

Compute node hardware is a zoo:

- Vastly different numbers of cores
- Vastly different memory architectures
- Accelerator cards: GPGPUs, Xeon Phi
- Heterogeneous chip architectures: AMD Fusion

Computing in the Age of Multi- and Many-Core

Compute node hardware is a zoo:

- Vastly different numbers of cores
- Vastly different memory architectures
- Accelerator cards: GPGPUs, Xeon Phi
- Heterogeneous chip architectures: AMD Fusion

Programming diverse hardware is uneconomic:

- Various low-level programming models
- Each requires different expert knowledge
- Heterogeneous combinations of the above ?
- Cumbersome, error-prone and inefficient

An Alternative: Single Assignment C (SAC)

Credo: abstraction, abstraction, abstraction

- Program what to compute, not exactly how
- Leave concrete organisation of execution to compiler and runtime system
- Put expert knowledge into tools, not into applications
- **Architecture-agnostic** programming for portability
- Compile one source to diverse target hardware
- Automatically manage resources: memory, cores, etc

SAC — Design Space

$\boxed{\textbf{SAC}}$

SAC — Design Space

High-level functional, data-parallel programming with vectors, matrices, arrays

SAC

SAC — Design Space

High-level functional, data-parallel
programming with vectors, matrices, arrays

SAC

Suitability to achieve high performance
in sequential and parallel execution

SAC — Design Space

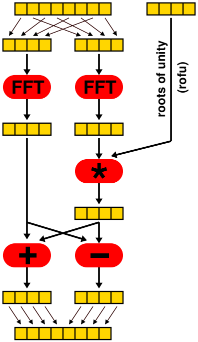
High-level functional, data-parallel programming with vectors, matrices, arrays

SAC

Easy to adopt for programmers with an imperative background

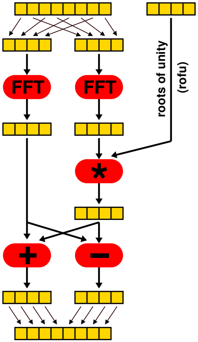
Suitability to achieve high performance in sequential and parallel execution

Case Study: 1-Dimensional Complex FFT (NAS-FT)




```
complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even = condense(2, v);
  odd  = condense(2, drop( [1], v));

  even = FFT( even, rofu);
  odd  = FFT( odd,  rofu);

  rofu = condense( len(rofu) / len(odd), rofu);

  left  = even + odd * rofu;
  right = even - odd * rofu;

  return left ++ right;
}
```

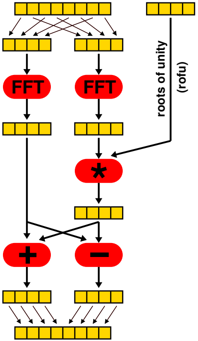
Case Study: 1-Dimensional Complex FFT (NAS-FT)




```
complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even = condense(2, v);
  odd  = condense(2, drop( [1], v));

  even = FFT( even, rofu);
  odd  = FFT( odd,  rofu);

  rofu = condense( len(rofu) / len(odd), rofu);

  left  = even + odd * rofu;
  right = even - odd * rofu;

  return left ++ right;
}
```

Now, what is functional array programming ?

Functional Array Programming

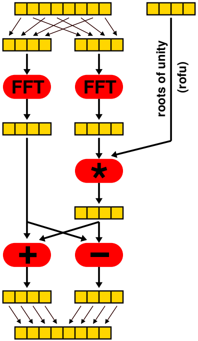



```
complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even = condense(2, v);
  odd  = condense(2, drop( [1], v));

  even = FFT( even, rofu);
  odd  = FFT( odd,  rofu);

  rofu = condense( len(rofu) / len(odd), rofu);

  left  = even + odd * rofu;
  right = even - odd * rofu;

  return left ++ right;
}
```

Role of Functions:

- Map argument values to result values
- No side effects
- Call-by-value parameter passing

Functional Array Programming

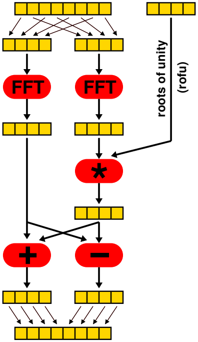



```
complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even = condense(2, v);
  odd  = condense(2, drop( [1], v));

  even = FFT( even, rofu);
  odd  = FFT( odd,  rofu);

  rofu = condense( len(rofu) / len(odd), rofu);

  left  = even + odd * rofu;
  right = even - odd * rofu;

  return left ++ right;
}
```

Role of Variables:

- Variables are placeholders for values
- Variables do not denote memory locations
- Automatic memory management

Functional Array Programming

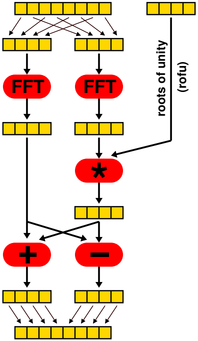



```
complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even = condense(2, v);
  odd  = condense(2, drop( [1], v));

  even = FFT( even, rofu);
  odd  = FFT( odd,  rofu);

  rofu = condense( len(rofu) / len(odd), rofu);

  left  = even + odd * rofu;
  right = even - odd * rofu;

  return left ++ right;
}
```

Execution Model:

- Contextfree substitution of expressions
- No side-effects

Functional Array Programming

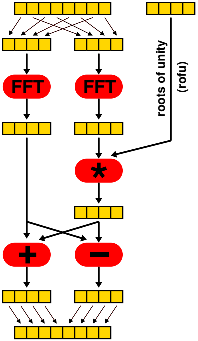



```
complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even = condense(2, v);
  odd  = condense(2, drop( [1], v));

  even = FFT( even, rofu);
  odd  = FFT( odd,  rofu);

  rofu = condense( len(rofu) / len(odd), rofu);

  left  = even + odd * rofu;
  right = even - odd * rofu;

  return left ++ right;
}
```

Control flow constructs:

- Branches are syntactic sugar for conditional expressions
- Loops are syntactic sugar for tail-end recursive functions
- Data flow determines execution order

Functional Array Programming



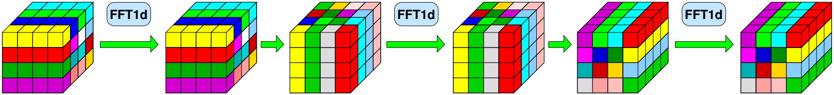

```
complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even = condense(2, v);
  odd  = condense(2, drop( [1], v));

  even = FFT( even, rofu);
  odd  = FFT( odd,  rofu);

  rofu = condense( len(rofu) / len(odd), rofu);

  left  = even + odd * rofu;
  right = even - odd * rofu;

  return left ++ right;
}
```

Nature of Arrays:

- Pure values, mapping indices to (other) values
- No state, no fixed memory representation
- Maybe no memory manifestation at all

Case Study: 3-Dimensional Complex FFT (NAS-FT)

Algorithmic idea:



Implementation:

```
complex[.,.,.] FFT( complex[.,.,.] a, complex[.] rofu)
{
  b = { [.,y,z] -> FFT( a[.,y,z], rofu) };
  c = { [x,.,z] -> FFT( b[x,.,z], rofu) };
  d = { [x,y,.] -> FFT( c[x,y,.], rofu) };

  return d;
}

typedef double[2] complex;
```

The Same in Fortran

```fortran
subroutine fft(dir, x1, x2)
implicit none
include 'global.h'
integer dir
double complex x1(ntotal), x2(ntotal)
double complex scratch(fftblockpad_default
>                      *maxdim*2)
if (dir .eq. 1) then
   call cffts1(1, dims(1,1), x1, x1, scratch)
   call cffts2(1, dims(1,2), x1, x1, scratch)
   call cffts3(1, dims(1,3), x1, x2, scratch)
else
   call cffts3(-1, dims(1,3), x1, x1, scratch)
   call cffts2(-1, dims(1,2), x1, x1, scratch)
   call cffts1(-1, dims(1,1), x1, x2, scratch)
endif
return
end
subroutine cffts1(is, d, x, xout, y)
implicit none
include 'global.h'
integer is, d(3), logd(3)
double complex x(d(1),d(2),d(3))
double complex xout(d(1),d(2),d(3))
double complex y(fftblockpad, d(1), 2)
integer i, j, k, jj
do i = 1, 3
   logd(i) = ilog2(d(i))
end do
do k = 1, d(3)
   do jj = 0, d(2) - fftblock, fftblock
      do j = 1, fftblock
         do i = 1, d(1)
            y(j,i,1) = x(i,j+jj,k)
         enddo
      enddo
```

```fortran
      call cfftz (is, logd(1),
>                 d(1), y, y(1,1,2))
      do j = 1, fftblock
         do i = 1, d(1)
            xout(i,j+jj,k) = y(j,i,1)
         enddo
      enddo
   enddo
enddo
return
end
subroutine cffts2(is, d, x, xout, y)
implicit none
include 'global.h'
integer is, d(3), logd(3)
double complex x(d(1),d(2),d(3))
double complex xout(d(1),d(2),d(3))
double complex y(fftblockpad, d(2), 2)
integer i, j, k, ii
do i = 1, 3
   logd(i) = ilog2(d(i))
end do
do k = 1, d(3)
   do ii = 0, d(1) - fftblock, fftblock
      do j = 1, d(2)
         do i = 1, fftblock
            y(i,j,1) = x(i+ii,j,k)
         enddo
      enddo
      call cfftz (is, logd(2),
>                 d(2), y, y(1, 1, 2))
      do j = 1, d(2)
         do i = 1, fftblock
            xout(i+ii,j,k) = y(i,j,1)
         enddo
      enddo
   enddo
enddo
return
end
```

```fortran
subroutine cffts3(is, d, x, xout, y)
implicit none
include 'global.h'
integer is, d(3), logd(3)
double complex x(d(1),d(2),d(3))
double complex xout(d(1),d(2),d(3))
double complex y(fftblockpad, d(3), 2)
integer i, j, k, ii
do i = 1, 3
   logd(i) = ilog2(d(i))
end do
do j = 1, d(2)
   do ii = 0, d(1) - fftblock, fftblock
      do k = 1, d(3)
         do i = 1, fftblock
            y(i,k,1) = x(i+ii,j,k)
         enddo
      enddo
      call cfftz (is, logd(3),
>                 d(3), y, y(1, 1, 2))
      do k = 1, d(3)
         do i = 1, fftblock
            xout(i+ii,j,k) = y(i,k,1)
         enddo
      enddo
   enddo
enddo
return
end
subroutine cfftz (is, m, n, x, y)
implicit none
include 'global.h'
integer is,m,n,i,j,l,mx
double complex x, y
dimension x(fftblockpad,n), y(fftblockpad,n)
mx = u(1)
do l = 1, m, 2
   call fftz2 (is, l, m, n, fftblock,
>              fftblockpad, u, x, y)
```

```fortran
   if (l .eq. m) goto 160
   call fftz2 (is, l + 1, m, n, ff
>              fftblockpad, u, y,
enddo
goto 180
160 do j = 1, n
   do i = 1, fftblock
      x(i,j) = y(i,j)
   enddo
enddo
180 continue
return
end
subroutine fftz2 (is, l, m, n, ny
implicit none
integer is,k,l,m,n,ny,ny1,n1,li,l
double complex u,x,y,u1,x11,x21
dimension u(n), x(ny1,n), y(ny1,n
n1 = n / 2
lk = 2 ** (l - 1)
li = 2 ** (m - 1)
lj = 2 * lk
ku = li + 1
do i = 0, li - 1
   i11 = i * lk + 1
   i12 = i11 + n1
   i21 = i * lj + 1
   i22 = i21 + lk
   if (is .ge. 1) then
      u1 = u(ku+i)
   else
      u1 = dconjg (u(ku+i))
   endif
   do k = 0, lk - 1
      do j = 1, ny
         x11 = x(j,i11+k)
         x21 = x(j,i12+k)
         y(j,i21+k) = x11 + x21
         y(j,i22+k) = u1 * (x11 - x2
      enddo
   enddo
enddo
return
end
```

The Power of Abstraction

- Programming by composition of building blocks:
  - Rapid prototyping
  - Good readability of code
  - High confidence in correctness
  - Plenty of code reuse opportunities

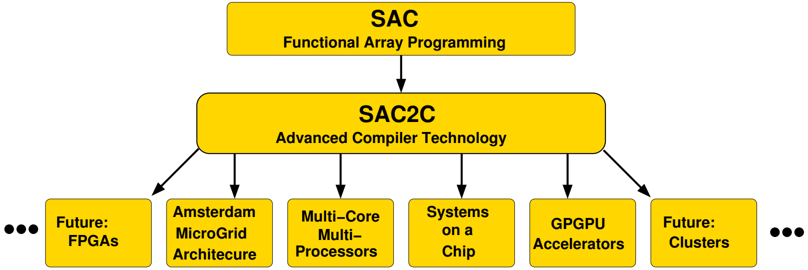
The Power of Abstraction

- Programming by composition of building blocks:
  - Rapid prototyping
  - Good readability of code
  - High confidence in correctness
  - Plenty of code reuse opportunities
- Opportunities for compiler and runtime system:
  - Target-independent optimisation
  - Code generation for variety of target architectures
  - Automatic parallelisation
  - Automatic memory management

The Power of Abstraction

- Programming by composition of building blocks:
  - Rapid prototyping
  - Good readability of code
  - High confidence in correctness
  - Plenty of code reuse opportunities
- Opportunities for compiler and runtime system:
  - Target-independent optimisation
  - Code generation for variety of target architectures
  - Automatic parallelisation
  - Automatic memory management
- Result:
  - Sufficient performance
  - For a range of architectures
  - Without extra effort

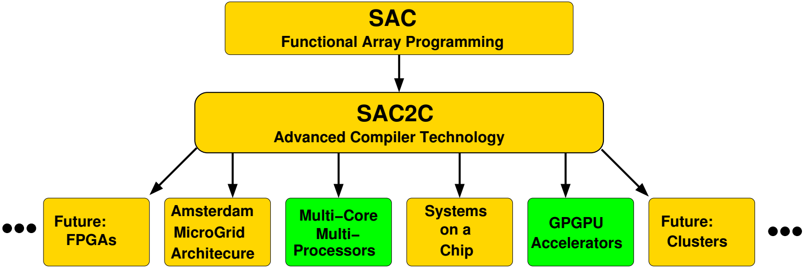
Compilation Challenge



And achieve reasonable performance....

Goal: achieve 90% with 10% of the effort

Compilation Challenge: Heterogeneous Architectures



Simultaneously use:

- Multiple CPU processors with multiple cores
- Multiple GPU accelerators

SAC Array Comprehensions: With-loops

Internal representation of any array operation:

```
A = with {
      ( lower_bound <= idxvec < upper_bound ) : expr ;
       ...
      ( lower_bound <= idxvec < upper_bound ) : expr ;
    }: genarray( shape , default )
```

Characteristics:

- Disjoint partitions defined by *generators*
- Induction variable: *idxvec*
- Partitions associated with expression
- Periodically sliced and interleaved partitions

Parallelisation of With-Loops

General approach:

- **CPU:** Offload model with additional worker threads
- **GPU:** Offload model with dedicated kernel

Parallelisation of With-Loops

General approach:

- **CPU:** Offload model with additional worker threads
- **GPU:** Offload model with dedicated kernel

Compilation process:

- **CPU:** Intertwine generators for canonical order processing
- **GPU:** Turn each generator into one kernel

Parallelisation of With-Loops

General approach:

- **CPU:** Offload model with additional worker threads
- **GPU:** Offload model with dedicated kernel

Compilation process:

- **CPU:** Intertwine generators for canonical order processing
- **GPU:** Turn each generator into one kernel

Solution for heterogeneous architectures:

- Follow both compilation paths at the same time
- Maintain two different versions of each with-loop

Host vs Device Memories

Existing compilation paths:

- **CPU:** everything in host memory
- **GPU:** whole arrays either in host or device memory
- **GPU:** automatic transfers between host and device memory

Host vs Device Memories

Existing compilation paths:

- **CPU:** everything in host memory
- **GPU:** whole arrays either in host or device memory
- **GPU:** automatic transfers between host and device memory

Heterogeneous computing:

- **Distributed arrays** partially stored in each memory
- Control structure to track what data is available where
- Automatic transfers of array slices as necessary
- Decision when transfer is made at runtime
- Caching of array slices in multiple memories

Runtime Organisation

Host threads:

- 1 master thread executes sequential parts of code on host
- k worker threads run parallel computations offloaded to host
- 1 controller thread per accelerator controls computations offloaded to that accelerator

Runtime Organisation

Host threads:

- 1 master thread executes sequential parts of code on host
- k worker threads run parallel computations offloaded to host
- 1 controller thread per accelerator controls computations offloaded to that accelerator

Scheduling:

- Existing scheduling techniques divide iteration space between threads: static, dynamic, etc

Runtime Organisation

Host threads:

- 1 master thread executes sequential parts of code on host
- k worker threads run parallel computations offloaded to host
- 1 controller thread per accelerator controls computations offloaded to that accelerator

Scheduling:

- Existing scheduling techniques divide iteration space between threads: static, dynamic, etc
- Observation: GPU threads generally compute much faster than CPU threads
- Idea: monitor and statistically analyse relative performance between GPUs and CPUs, then divide iteration space accordingly
- Use dynamic scheduling within peer groups
- Implementation now: choose for experimentation

Experimental Evaluation

Experimental Setting:

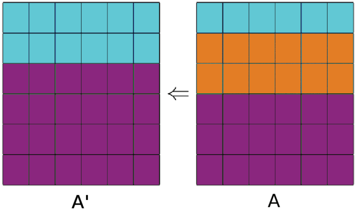
- DAS-4 cluster
- Dual-GPGPU Machine:
  - 2 hexa-core 2.67 GHz CPUs
  - 2 NVidia GTX480 GPGPUs
- Multi-GPGPU Machine:
  - 2 quad-core 2.4 GHz CPUs
  - 8 NVidia GTX580 GPGPUs

Example: Relaxation with Fixed Boundary Conditions

Main computation in intermediate representation:

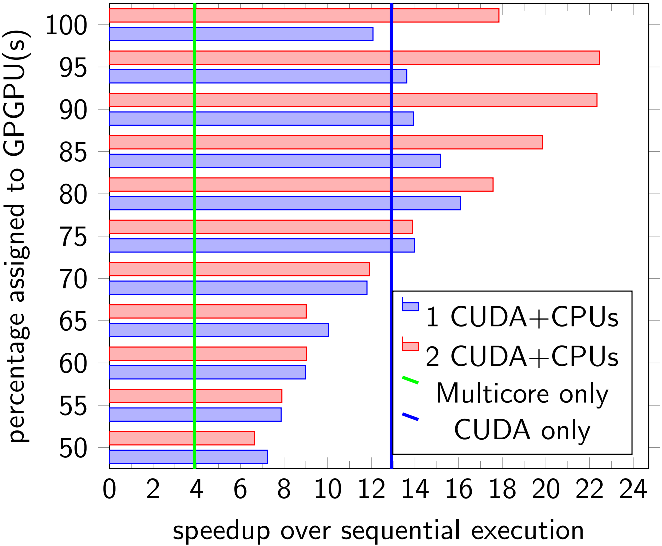
```
for (i=0; i<iter; i++) {
  A = with {
        ([0,0] <= x < shape(A)) :
                0.25 * ( A[x+[1,0]] + A[x-[1,0]]
                       + A[x+[0,1]] + A[x-[0,1]]);
     } : modarray( A);
}
```

Memory organisation:



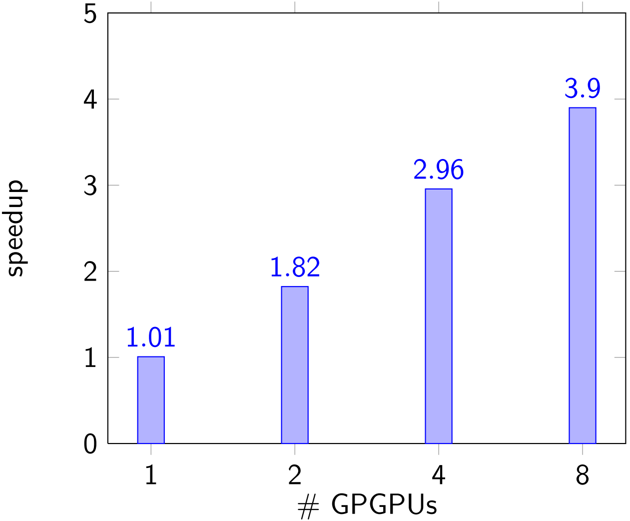
Relaxation with Fixed Boundary Conditions

Dual-GPGPU machine, 6000x6000 matrix, 2000 iterations.

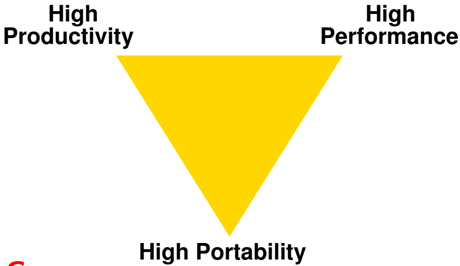


Relaxation with Fixed Boundary Conditions

Multi-GPGPU machine, 8000x8000 matrix, 10000 iterations.



Conclusion

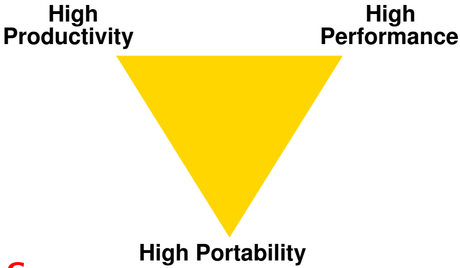


Single Assignment C:

- reconcile productivity, portability and performance
- one source for all architectures
- exposure of fine-grained concurrency
- automatically sequentialising compiler
- automatic resource management
- **automatic exploitation of heterogeneous architectures**

www.sac-home.org

Conclusion

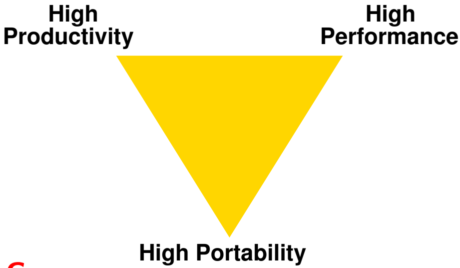


Single Assignment C:

- reconcile productivity, portability and performance
- one source for all architectures
- exposure of fine-grained concurrency
- automatically sequentialising compiler
- automatic resource management
- **automatic exploitation of heterogeneous architectures**
- Is it worth it ?

www.sac-home.org

Conclusion



Single Assignment C:

- reconcile productivity, portability and performance
- one source for all architectures
- exposure of fine-grained concurrency
- automatically sequentialising compiler
- automatic resource management
- **automatic exploitation of heterogeneous architectures**
- Is it worth it ?
- Future work: expand distributed arrays to clusters

www.sac-home.org